A Model Interpreter for Timed Automata

M. Usman Iftikhar[1], Jonas Lundberg[1], Danny Weyns[2]

[1] Institute of Computer Science,
Linnaeus University, 351 95 Växjö, Sweden,
usman.iftikhar@lnu.se
[2] Katholieke Universiteit Leuven, Belgium
Linnaeus University, Växjö, Sweden
danny.weyns@kuleuven.be

Abstract. In the model-centric approach to model-driven development, the models used are sufficiently detailed to be executed. Being able to execute the model directly, without any intermediate model-to-code translation, has a number of advantages. The model is always up-to-date and runtime updates of the model are possible. This paper presents a model interpreter for timed automata, a formalism often used for modeling and verification of real-time systems. The model interpreter supports real-time system features like simultaneous execution, system wide signals, a ticking clock, and time constraints. Many existing formal representations can be verified, and many existing DSMLs can be executed. It is the combination of being both verifiable and executable that makes our approach rather unique.

Keywords: model-driven development, model interpretation, timed automata, virtual machine

1 Introduction

Model-driven development (MDD) is a software development methodology focusing on creating and exploiting domain models [17]. A domain model is an abstraction that describes selected aspects of a specific domain. An important part of MDD is the use of domain-specific modeling languages (DSML) [6]. Developers use DSMLs to efficiently build application models using elements of the domain and often express design intent declaratively rather than imperatively.

In a model-centric approach, models of the system are established in sufficient detail that the model can be executed, or used to generate executable code [17]. To achieve this, the models defined in a DSML might include, for example, representations of persistent and non-persistent data, business logic, and presentation elements. Integration to legacy data and services might require that the interfaces to those models are also modeled.

There are two common approaches to model execution. In the code-generation approach a DSML specified model can be translated to a program in a language like Java that can later be executed using the standard Java virtual machine. This approach works fine when the DSML is (roughly) a more abstract, richer

version of an ordinary programming language. However, code-generation runs into trouble when the model has more declarative features like *simultaneous execution*, *system wide signals*, and *time constraints*, that is, model features that have no simple counterpart in the target language into which it should be translated. Furthermore, model updates at runtime are basically impossible and any manual change in the generated code will ruin the connection to the model.

An alternative to code-generation is model interpretation that relies on the existence of a virtual machine able to directly read and run the model. The major advantage of this approach is that model updates at runtime are possible (see Section 5) and, as we will see in Section 3 and 4, the domain specific interpreter can provide support for model specific declarative features like the ones presented above. Having the model available at runtime also simplifies runtime verification of model dependent system goals (see Section 5).

The goal of this paper is to present a model interpreter for timed automata [2], first presented in [10]. Timed automata are an often used formalism to model real-time systems and it supports features like simultaneous execution, system wide signals, and time constraints[3]. Timed automata has a graphical representation suitable for humans and a corresponding XML based DSML suitable for machine processing. Formal properties (system goals) of models described by timed automata can be verified by a tool called Uppaal [4]. In addition to handling real-time features, it is the use of a domain specific model being verifiable, executable in a real world scenario, and allowing model updates at runtime that makes our approach rather unique. See related work in Section 6 for more details.

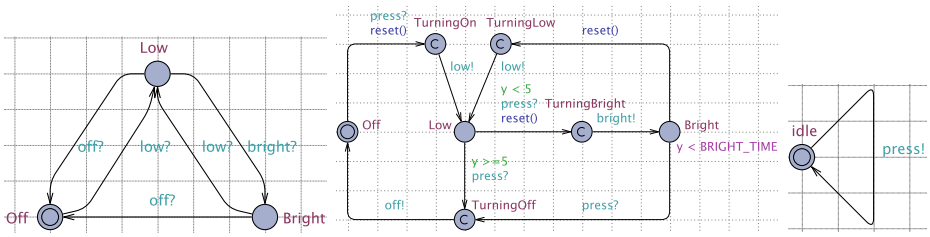
Timed automata will be presented in Section 2. The model interpretation is done in two steps: 1) The DSML defining the model is translated into an internal task graph based executable model (Section 3), and 2) A virtual machine, specifically designed for timed automata, interprets the executable model (Section 4). Step 1) is not novel since standard techniques from compiler design are used. The virtual machine on the other hand has novel features extending the functionality of a standard stack machine to handle a wide set of timed automata specific features. Additional features of our approach (e.g. support for runtime model updates and runtime verification) are discussed in Section 5. In Section 6 we present related work, and in Section 7 we present summary and conclusions.

2 Timed Automata

A timed automaton [2] is a finite automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, all clock values increase with the same speed. The clock values can be compared to integers and these comparisons form guards that may enable or disable transitions and therefore constrain the automaton's behavior.

Uppaal [5] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. Up-

[3] See the `uppaal.org` website for a list of industrial projects using timed automata and the Uppaal verification tool.



(a) Lamp (b) Lamp Controller (c) User

Fig. 1: The simple lamp example.

PAAL comes with an XML based description language in which systems of timed automata can be defined, which is our DSML. UPPAAL also includes a number of tools for visualizing the automata, simulation, and model verification. The aim of this section is to provide a brief introduction to timed automata as defined by the UPPAAL DSML. It can be considered as a brief (and informal) summary of the official UPPAAL tutorial [4], inspired by [8], with a focus on modeling and interpretation of timed automata. To simplify the presentation we use standard automata terminology (e.g. state, transition) rather than the standard timed automata terminology (e.g. location, fire an edge).

2.1 Networks of Timed Automata

A timed automaton is a finite-state machine extended with clock variables. All clocks progress synchronously. In UPPAAL, a system is modelled as a network of several such timed automata in parallel. The model is further extended with ordinary variables and the state of the system is defined by the state of all automata, the clock values, and the values of the variables. An automata may make a state transition separately or due to synchronization with another automata through channels. For example, for a channel x, a sender $x!$ can synchronize with a receiver $x?$ through a signal.

Figure 1 shows three automata modelling a simple system with a lamp, a lamp controller, and a button to be pressed by a user. At start, when both the lamp and the controller are in state *Off*, if the user presses a button a signal *press!* is sent and the controller moves to state *TurningOn* due to synchronization *press?* followed by *LowLight* (sending a signal *low!*), and the lamp is turned on (due to *low?*). If the user presses the button again, the lamp is turned off. However, if the user is fast and within 5 time units presses the button twice, the lamp is turned on and becomes bright. The clock y of the lamp controller is used to detect if the user was fast ($y < 5$) or slow ($y >= 5$). The lamp stays bright for a certain period of time *BRIGHT_TIME* and then returns to *Low* state again.

We divide the models into two categories: *environment models* and *system models*. Environment models are used for simulation and enables offline verification of the system by providing input and getting output. For example, the user and lamp models are environment models in our lamp example. The system

models are the models that contain the actual domain functionality/logic. In our lamp example, the lamp controller model is the system model.

The edges of the automata are annotated with three types of labels: a *guard*, expressing a condition (e.g. $y < 5$) on the values of clocks and variables that must be satisfied for the edge to be taken; a *synchronization* action (e.g. *press!*) which, when the edge is taken, forces a synchronization with other components on a complementary action, and an *update* (e.g. the function call *reset*() which resets clock y to 0) defining actions to be taken when a transition is made. All three types of labels are optional: absence of a guard is interpreted as the condition *true*, and absence of a synchronization action indicates an internal (non-synchronizing) edge (e.g. *BrightLight* $\rightarrow$ *TurningLow* in the controller).

Only one state per automaton, called *control* or *active* state, is active at a time. States can also be annotated with *invariants* expressing constraints on the clock values for control to remain in a particular state. For example, the system can only remain in *BrightLight* as long as the value of y is less than *BRIGHT_TIME*.

UPPAAL defines two types of transitions between states: *action transition* and *delayed transition*. Action transitions can be further divided into *synchronization transition* and *internal transition*. If two complementary labeled edges (e.g. *press!* and *press?*) in two different automata are enabled then they can synchronize and a simultaneous *synchronization transition* is activated. In a delayed transition only the clock ticks and no actual state transition is made (e.g. *Bright* remains active in the controller while $y <$ *BRIGHT_TIME* and as long as no-one is pushing the button). Further progress in time might lead to an invariant violation ($y \geq$ *BRIGHT_TIME*) and an *internal transition* (*Bright* $\rightarrow$ *TurningLow*).

Finally, to enable modeling of atomicity of transition sequences in a given automaton (i.e. multiple transitions with no time delay) states may be marked as *committed* (indicated by a *c* in the circle). Commited states (e.g. *TurningOn* in the controller) make it possible to receive a signal (*press?* in *Off* $\rightarrow$ *TurningOn*) and send a signal (*low!* in *TurningOn* $\rightarrow$ *LowLight*) without any time delay.

2.2 The Timed Automata Modeling Language

The timed automata DSML is a straight forward XML markup of the transition graphs described previously. States (locations in UPPAAL) are nodes with a number of attributes (`id`, `name`, `commited`, `invariant`, etc.) and transitions are edges connecting source and target states (identified by their `id`s) with attributes (`guard`, `synchronization`, `assignment`) describing the transition conditions.

Figure 2 shows an excerpt of the DSML for our lamp example. It starts with a section of global declarations `<declaration>` with variables and signals that are accessible anywhere in the system. In our lamp example, the global declaration section consists only of signal declarations, i.e., *press*, *off*, *low*, and *bright*.

The declaration section is followed by one or more template definitions describing a single automaton. Templates have a name (element `<name>`), a set of local variables and clocks (`<declaration>`), a set of states (`<location>`), an initial state (element `<init>`), and a set of transitions (`<transition>`).

```
<?xml version="1.0" encoding="utf-8"?>
<nta>
    <declaration>chan press, off, low, bright;</declaration>
    <template>
        <name>LampController</name>
        <declaration>const int BRIGHT_TIME = 10;
            clock y;
            void reset(){ y = 0;}
        </declaration>
        <location id="id0">
            <name>TurningLow</name>
            <committed/>
        </location>
        <location id="id1">
            <name>Bright</name>
            <label kind="invariant">y &lt; BRIGHT_TIME</label>
        </location>
        <location id="id2">
            <name>Off</name>
        </location>
        <init ref="id2"/>
        <transition>
            <source ref="id1"/>
            <target ref="id0"/>
            <label kind="synchronisation">low!</label>
            <label kind="assignment">reset()</label>
        </transition>
        <!-- missing transitions -->
    </template>
    <!-- missing templates for Lamp and User-->
    <system> system Lamp, LampController, User;</system>
</nta>
```

Fig. 2: Excerpt of the XML based DSML for the Lamp example.

The final DSML section, the *system declaration*, lists the *automata instances* planned to be used in the system. The system section is a description of how the system is going to be initialized. In our lamp example, we have one instance of each of the *User*, *Lamp* and *LampController* automata. In general however, a system might contain multiple instances (e.g. multiple users) of a single automaton.

3 Executable Model Generation

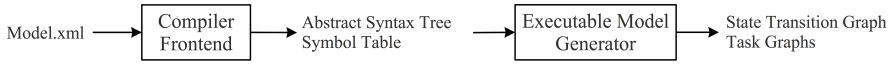


Fig. 3: Overview of the executable model generation.

A network of timed automata as described in Section 2 is a system model with sufficient detail to be interpreted. The model interpretation can be divided into two steps: 1) Executable Model Generation, and 2) Model Execution. Both are handled in sequence each time a model is executed. In this section we present the model generation and in Section 4 we present the model execution.

An overview of the executable model generation is presented in Figure 3. The input is an XML based DSML describing the system (`Model.xml`), the final result (`State Transition Graphs`, `Task Graphs`) is an internal representation of the system that later will be executed by our virtual machine. The executable

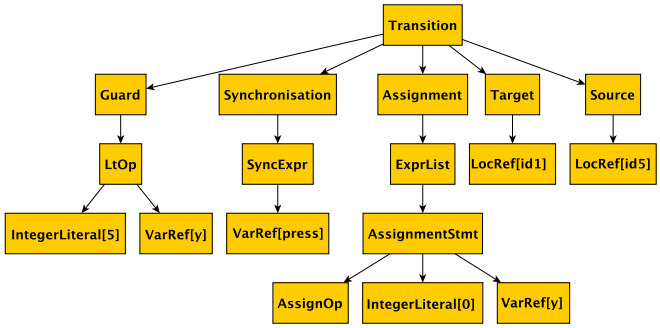


Fig. 4: AST for transition *Low* to *TurningBright*.

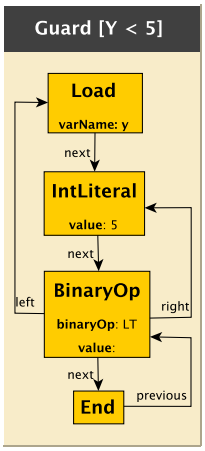


Fig. 5: Task graph for guard y < 5

model generation is divided into two steps: 1) A *compiler frontend* that parses the input XML file and creates a single abstract syntax tree (AST) and a symbol table. 2) An *executable model generator* that traverses the AST to generate the final executable model representation.

The compiler frontend uses standard compiler techniques and will not be described in detail. In short, UPPAAL DSML is defined as a context-free grammar that can easily be used to generate a parser using the Antlr [16] parser generator tool. The resulting AST is then traversed once more to construct a symbol table, a mapping from scopes to variable declarations. A scope in the UPPAAL DSML can be a global declaration, system declaration, template declaration, or a function.

Figure 4 shows a subtree of the AST representing the transition *Low* to *TurningBright* in LampController. Apart from source and target information of the transition (`Source`, `Target` subtrees) it also includes three labels: `Guard` ($y < 5$), `Synchronization` (*press*?), and `Assignment` ($y = 0$).

The executable model representation later to be executed by a virtual machine consists of two parts: 1) *State transition graphs*, one for each automata, and 2) a number of *Task graphs*. The state transition graphs are just an internal graph representation of the system's timed automata as described in Section 2. There exists one graph for each automaton. The states are nodes and the transitions are edges. Both nodes and edges are annotated with references to task graphs. Each transition label (guard, synchronization, update) is represented by a separate task graph, and each node attribute (invariant) is also represented as a task graph.

A task graph defines the control flow of a task graph evaluation. It consists of a collection of task nodes that are connected with *next* and *previous* attributes. Each task node has a task type attribute defining the role of that node. Examples of task types are: *DECL* declares a variable, *LITERAL* defines a integer literal, *BINARY_OP* for binary operations, $STORE/LOAD$ store/load a variable value from/to the heap, *END* signals the termination of a task graph evaluation, etc. Depending upon task type a node can have additional attributes, e.g. the task node for binary operators have *left* and *right* attributes pointing to left and right expression nodes.

Fig. 5 shows the task graph for the guard $y < 5$ of the Low to $TurningBright$ transition in Lamp Controller. The execution order is defined by the *next* edges (non-essential *previous* edges are omitted for simplicity) and the less-then operator is represented as a binary operation (tagged with LT) with two node type specific edges (*left* and *right*) referencing the values to be used in the operator. The *previous* edge in the *END* node points to the final result of the task graph evaluation.

In addition to task graphs generated due to various state and transition attributes we also generate task graphs for all declarations of global variables and signals defined in the `<declaration>` part of the AST, and for all clock and variable declarations local to a certain automaton. These additional task graphs are not directly referenced by any transition graph, they will be used in the initialization phase of the virtual machine before the execution starts.

Task and transition graphs are generated in a single AST traversal. Due to space limitations the actually used algorithm will not be presented here.

4 Model Execution

The model interpretation starts with an initialization phase (Section 4.1) where global and template variables are declared and initialized, the real-time time unit is set, and connections to environment models are established. Then the actual execution can start (Section 4.2).

The core of the model interpreter is the *timed automata virtual machine* (TAVM). Apart from heap and stack management the TAVM has two parts that together are responsible for the actual execution. The *state transition machine* (STM) is responsible for the state transitions, and the *task graph interpreter* (TGI) is (on requests from the STM) evaluating task graphs. Several of the design decisions for the TAVM are inspired by Uppaal Tron [9], a model based testing tool from Uppaal.

4.1 Virtual Machine Setup

Declarations: The first step is to execute global and system declarations by the task graph interpreter in order to initialize all variable and clock declarations used by the system. For example, it declares what channels are going to be used. The system declarations provide a list of automata instances that are to

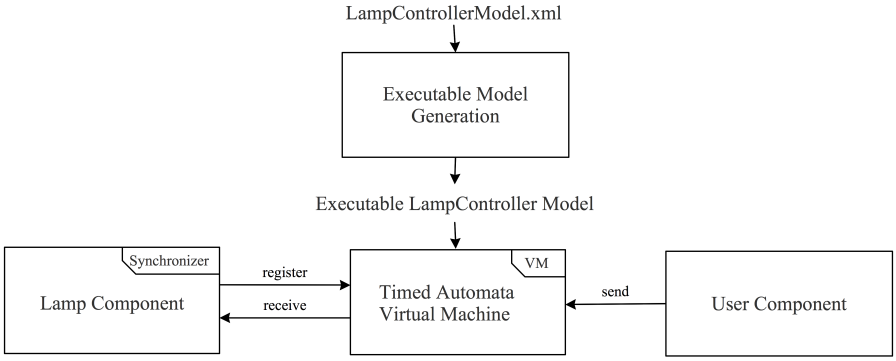


Fig. 6: Overview of the Lamp example interpretation.

be executed by the virtual machine. Finally, for each instantiated automaton, all local declarations are executed and a list of initial active states is created.

Model time unit: In timed automata, a time tick is an abstract entity that can be assigned to any real time unit, e.g. milliseconds, seconds, minutes, etc. In order to correctly behave as real-time clocks, the TAVM must know the real time unit of a tick. It therefore provides a method $setRealTimeUnit(milliseconds)$ to set the time unit in milliseconds. How clocks progress is discussed in more detail at Section 4.2.

Environment connection: As mentioned in Section 2.2, the entire model is divided into two categories: Environment models representing external components that interact with the running system, and system models that are to be executed in the TAVM. The TAVM connects with the environment through signals defined in the automata model.

Figure 6 shows an overview of the Lamp example interpretation. The input is an XML based system specification (LampControllerModel.xml) that is used to generate an executable model which is then fed to the TAVM for execution. At runtime, TAVM must be connected to a real lamp and a real button. To realize this in our approach, we replace the models of the environment with an actual environment represented by the Lamp and User components in Figure 6, and the TAVM executes only the Lamp Controller model. A component in this case is a piece of software which handles the communication with external devices.

The TAVM assigns a unique identifier to each channel. This identifier can be used to send and receive signals from the TAVM. TAVM has a public interface (named VM) providing a method $getChannelId(\text{“}channel\text{”})$ that can be used to get channel identifiers. To send a signal, the VM interface provides a $send(channelId)$ method that can be used to send a signal from the environment to the virtual machine. Data can also be sent using the *send* method as a string expression like “$a = 2$”. These expressions are converted to task graphs on-the-

fly and evaluated by the task graph interpreter when a signal is consumed. More details about how signals are consumed are provided in the next section.

The TAVM provides an abstract class *Synchronizer*, that should be extended by components interested in receiving signals from the TAVM. A component registers itself for a certain channel by, first, getting the channel identifier using the *getChannelId* method, and then call the *register* method provided by the VM interface. The *register* method take three parameters: 1) a channel identifier identifying which type of signal we are interested in, 2) an instantiation of the *Synchronizer* class, that will receive the signals, 3) and a array of variable names specifying what variable values we are interested in. The *Synchronizer* class defines one abstract method *receive* that has two parameters: 1) a channel identifier that can be used to determine which signal is received, 2) the data that comes with the signal.

4.2 Virtual Machine Execution

The TAVM provides a *start* method which starts the actual execution once the setup is completed. Once started, the virtual machine is idle until triggered either by input from the environment, or by a time tick. The heart of the TAVM is the State Transition Machine (STM). The STM keeps track of all active nodes and decides what and when transitions are triggered. The STM is using another component, the Task Graph Interpreter (TGI), whenever a task graphs needs to be evaluated. In what follows we first present the STM and then the TGI.

The State Transition Machine. The STM maintains a set of all active nodes N and a set S, representing the current state, containing N and the values of all the variables and clocks. From now on "state" refers to the global state S and we refer to individual timed automata states/locations as nodes. Upon start, the STM checks all instantiated models and execute those that are in a committed state. To do that, the STM checks (one by one) all the active nodes in N, if a node is in a committed state, then STM randomly selects one outgoing transition from that node and tries to execute it. If that transition cannot be taken (e.g. a guard evaluates to false), STM tries another one. This process is repeated until all committed nodes are handled, and will also be repeated after each taken transition ending up in a committed state. The STM supports non-determinism by randomly selecting nodes and transitions if multiple available.

Algorithm 1 shows the pseudo code for executing one transition which does not interact with the environment. That is, it can only handle signals sent and received within the system model. The handling of signals involving external components is discussed later on.

In what follows, S' is a temporary state that can be rolled back to S, or S can become S', and if there is no guard on a transition (or invariant on a node), then the evaluation of the guard (invariant) expression returns true. Evaluation calls (e.g. $evaluateGuard(transition, S)$) are calls to the task graph interpreter requesting a task graph guard ($transition$) to be evaluated in a given state (S).

A1 Algorithm for executing a transition

Input N set of all active nodes
Input S current state including N and the value of all variables and clocks
Input *transition* to be taken and it's source *node*
Return *true* if transition accepted, otherwise $false$

```
1.  recvTransition ← NULL
2.  if evaluateGuard(transition, S) == true then
3.     if transition.synch != null and transition.synch.type == SEND then
4.        channelId = evaluateSynchronization(transition, S)
5.        recvTransition = findReceivingTransition(channelId, N, S)
6.        if evaluateGuard(recvTransition) != true then
7.           return false
8.        end if
9.     end if
10.    S' ← S
11.    evaluateUpdate(transition, S')
12.    if recvTransition! = null then
13.       evaluateUpdate(recvTransition, S')
14.    end if
15.    if checkAllInvariants(N, S') == true then
16.       S ← S'
17.       N.remove(node)
18.       N.add(transition.targetNode)
19.       if recvTransition != null then
20.          N.remove(recvTransition.srcNode)
21.          N.add(transition.targetNode)
22.       end if
23.       return true
24.    else
25.       discard(S')
26.       return false
27.    end if
28. end if
29. return false
```

Algorithm 1 starts by making sure that a transition can only be taken when the guard of the transition is true (line 2). If guard is true, and the transition involves synchronization (line 3), then it makes sure that the guard of the receiving transition is also true (line 6). After these preliminary checks we have a potential transition to a new state and we clone the current state ($S' \leftarrow S$, line 10) to make sure that we can roll back to S if future steps fails. Then we start to evaluate the update task graphs (line 11, 13), and verify that all invariants still holds (line 15). These steps might update S' and still fail. If they succeed we decide to make the transition and update the current state $S \leftarrow S'$ (line 16) and update N by adding and removing the old and new active node (also for the signal receiving transition), lines 17-22.

In order to communicate with the environment, we must modify our algorithm at a few places. For sending a signal to the environment, and after getting $channelId$ of the sender, we must look at the list of registered synchronizers. If any synchronizer is found registered for the same channel, we take the transition after executing update task graph and evaluating all the invariants. Then we call the *receive* method of the associated instance of the $Synchronizer$ class with the requested data.

When a signal is received from the environment, the STM finds the receiving transition through $channelId$, and execute the *guard* and *update* task graphs. It can happen that the system models in the STM and the environment models are not synchronized, and there is no transition at the moment who could receive the signal. STM then takes a flexible approach, and if the signal is not consumed, that signal is moved to a queue. Then the queue of signals is checked repeatedly whenever the clock ticks or a new signal arrives to consume the pending signals.

The STM maintains an internal timer, whose time period can be configured as discussed in Section 4.1. The STM keeps an internal data structure for all the clocks in the model. When the timer ticks, the STM temporarily increases the time of all the clock variables modelled in the automata, i.e, S' and checks the invariants of all enabled nodes. If all the invariants hold, the STM increases time for all the clock variables permanently $S \leftarrow S'$ and the timer goes to wait state. If the invariants of any active nodes are violated by the temporary increment of the timer, the STM reverts the time increment S and executes those nodes first, whose invariants are violated, by evaluating their transitions as discussed in the Algorithm 1. If a node can not take a transition, then the system ends in a timelock (this points to a design flaw in the model). The STM will then stop execution and throw a $TimelockException$.

If the selected time tick unit is very small we might end up in a situation where the TAVM can not manage all the required computations (or transitions) before the next tick. In addition to the general STM overhead this might occur when waiting for an external signal or due to certain time consuming TGI computations to check if a transition is possible or not. Our implementation handles this situation by buffering the time ticks and then executes them as soon as possible. This is (of course) problematic since it might cause a delay in the signals sent to the real world components. Thus, for each application, the real time unit to be used should be chosen carefully to make sure that the TAVM always manage to do all the required work before the next tick.

The Task Graph Interpreter. The task graph interpreter (TGI) evaluates task graphs on request from the STM. On initialization of the model, the STM requires the TGI to evaluate the initialization expressions for all the declared variables. Later on the TGI evaluates the *guard* and other transition and state attributes to take transitions as described previously in Algorithm 1. The TGI keeps track of a heap which stores all the global, system and template declarations, and a stack to store the state of local variables and function parameters when a call occur. Algorithm 2 shows an excerpt of the algorithm used by the

A2 Algorithm for task graph interpretation

Input *taskGraph* to be executed and the model instance identifier *processId*
Return Result of the task graph evaluation

```
1.  CT ← taskGraph.getFirst()
2.  while CT ∉ END do
3.     if CT ∈ LOAD then
4.        varName ← CT.getVarName()
5.        CT.value = heap.get(processId).get(varName)
6.     else if CT ∈ STORE then
7.        varName ← CT.getVarName()
8.        value ← CT.getPrev().getvalue()
9.        heap.get(processId).get(varName).setValue(value)
10.    else if CT ∈ BINARY_OP then
11.       op ← CT.getOp()
12.       if op ∈ LT then
13.          CT.value = CT.getLeft().getValue < CT.getRight().getValue()
14.       else
15.          ... more operators here
16.       end if
17.    else
18.       ... more tasks here
19.    end if
20.    CT ← CT.getNext()
21. end while
22. return CT.getPrev().getvalue()
```

TGI. With each evaluation request the STM also provides the *processId* that is needed to know which variables belongs to which instantiated model. For evaluating global and system declarations, the STM uses 0, and -1 respectively as *processId*. The CT (Current Task) always points to the current task. Upon receiving a request for evaluation, the TGI checks the task type (line 3, 6, 10) and takes the appropriate action. Once a task is evaluated, CT moves to the next task (line 20). This process is repeated until CT reaches the END task, which stops the task graph evaluation and the result of the evaluation is returned.

4.3 Validation

Apart from extensive in-house testing, our model interpreter has been evaluated in various adaptive systems. The original idea was presented in [10] where the adaptation logic of a robotic system is formally verified and executed by the model interpreter. Later on the model interpreter was evaluated in several case studies, including a smart house system, a security system, and two vehicular traffic systems [11]. Other applications where we applied the model interpreter are a digital story telling application and an e-health system [18]. See the project website [1] for more details about these case studies.

5 Additional Features and Future Work

Direct access to the model at runtime provides many additional advantages. Some of these features are already implemented and tested (Section 5.1) whereas others can be considered as future work (Section 5.2). See [10] for more details.

5.1 Additional Features

System Model Updates. The model interpreter supports online updates of the system models, which is crucial to deal with bugs, or adding new functionality to the running system. Our approach follows the classical process of runtime updates based on quiescence states [12]. The model interpreter provides a method $changeModel(model)$ which receives an updated model description (DSML). After that, the interpreter waits until each automaton of the current model reaches a quiescence state (i.e., no ongoing input or time triggered transactions) and interrupts the execution. The state of the current model is then saved and any new external inputs received while the update takes place are stored in a buffer. The interpreter then generates a new executable model (Section 3) and initialize that model (Section 4.1). Next, the interpreter restores the saved state of the previous model to the updated model and initializes new variables if applicable. Finally, the TAVM restarts the execution using the updated model.

Goal Verification. The model interpreter provides basic support for runtime verification of system goals. The *goal manager* component in the interpreter provides a function $addGoal(goal, client)$ that register goals to be monitored. A goal is a boolean expression involving clocks and variables (e.g. $y \leq 10$). The client is an implementation of the *GoalClient* interface registering to receive updates of the goal status. When a goal is registered the interpreter converts it to a task graph and start to notify the client every time a goal status is changed. Using this approach an interested component can track state changes and check whether the system goals hold or are violated. This feature was used in [10] to verify the correctness when updating the feedback loop models to deal with a new set of adaptation goals in a self-adaptive system.

Model Visualization. The model interpreter also comes with a graphical user interface allowing a user to inspect the running model, its ongoing execution, and to monitor variable values. This is useful for debugging the running system. The model interpreter provides a probe for interested components to get updates of the running model. The goal manager used in the goal verification uses the probe to listen to the updates and notifies the graphical user interface which display the current status of the model, see, e.g., Fig. 13-15 in [10].

5.2 Future Features

The goal manager currently used for both goal verification and model visualization has certain limitations. For example, goal types are limited to only boolean

expressions. In the future we plan to provide an interface offering plug and play facilities for arbitrary external components, and this new interface should give access to the complete model of the system (including the environment models) and allow every type of expression that can be represented as a task graph to be evaluated. This new machine interface opens up the possibility for a wide range of components to be attached to the virtual machine.

Our primary candidate for such plugin component is *online verification*. UPPAAL is foremost an offline verification tool. Given a model and a set of TCTL properties, the tool can prove that these properties are never invalidated. However, due to the so-called state explosion problem, incomplete knowledge about environment and memory constraints, offline verification may not be achieved. The interpreter on the other hand has runtime access to the complete model and can after each transition verify that the provided TCTL properties, converted to task graphs, are still valid. It is not a formal verification, it is however a pragmatic approach to verify that the running system behaves correctly. We are currently implementing an online verification component providing support for a subset of the timed computation tree logic (TCTL) properties, like constraints, safety and liveness properties.

Another possible approach to model checking problem is to delegate that work to other model checking tools. For example, using the plugin mechanism the model interpreter should be able to incorporate other trusted external modules (e.g., runtime model checking engines to support continuous verification at runtime).

6 Related Work

Ever since D.C. Schmidt's seminal paper on Model-Driven Software Engineering in 2006 [17] the interest for various aspects of model-driven design has flourished. In our approach we take the model-centric approach one step further and consider the model not only as a vehicle for code-generation, but also as a design specification suitable for verification. The number of existing models (DSMLs) that can be verified, executed in a real world environment, and that allows runtime model updates are rather few.

The Foundational Subset of Executable UML (fUML) defines the semantics for a subset of UML that can be executed by the fUML execution engine [15]. The fUML execution engine executes an in-memory representation of fUML models. Progress in the verification of these models has recently been achieved [14] but, to the best of our knowledge, no progress has been made yet for runtime model updates.

Ghezzi et al. [7] introduce adaptive model-driven execution to mitigate non-functional uncertainties. Using UML interaction diagrams a Markov decision model of the system is generated. The model is augmented with probability distribution of different execution paths of the system. The model is then executed by an ad-hoc interpreter that drives the execution of the system according to specified probabilities to guarantee the highest utility for a set of quality

properties. In their model each state is associated with an implementation of an abstract functionality of the system, and the interpreter invokes the implementations while state-by-state traversing the automaton, whereas we model and execute the actual implementation of the system. Markov decision models are well-known to allow probabilistic model checking and verification tools are available [13].

Anlauf et al. [3] presents an interpretable language XASM (Extensible Abstract State Machine). XASM uses a notion of external functions as defined in ASMs to realize a component-based modularization. The support environment of XASM consists of the XASM-compiler translating XASM programs to C source code, the runtime system, and the graphical debugging and animation tool. This approach lacks support for runtime update of the model, and although computer-aided verification of ASM models is possible in theory, it is well-known to be difficult in practice [19].

7 Summary and Conclusions

In this paper, we presented a model interpreter for timed automata, a formalism often used for modeling and verification of real-time systems. In addition to handling real-time features, it is the use of a domain specific model being verifiable, executable in a real world scenario, and allowing model updates at runtime that makes our approach rather unique. Given a model of the system the interpreter converts it into an executable model that can be interpreted by a timed automata virtual machine. Contrary to traditional approaches, where models are converted to code, using a model interpreter provides a number of additional advantages: 1) models are executed directly without converting them to a source code; hence no model-based testing is required, 2) models can be replaced at runtime without stopping the system, e.g., to add new functionality, 3) models can be used to verify system properties at runtime, 4) and it is also possible to visualize the running models. Our virtual machine can handle real-time system features like simultaneous execution, system wide signals, a ticking clock, and time constraints, not usually handled by ordinary stack based virtual machines. We included a future work section pointing out the possibility to use a model of the entire system to perform online verification.

A byte code version of the model interpreter can be downloaded from the project website [1].

References

1. ActivFORMS: Active Formal Models for Self-Adaptation. `http://homepage.lnu.se/staff/daweaa/ActivFORMS/ActivFORMS.htm`, 2016.
2. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, Apr. 1994.
3. M. Anlauff. *Abstract State Machines - Theory and Applications: International Workshop, ASM 2000 Monte Verità, Switzerland, March 19–24, 2000 Proceedings*,

chapter XASM- An Extensible, Component-Based Abstract State Machines Language, pages 69–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
4. G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
5. J. Bengtsson and W. Yi. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, chapter Timed Automata: Semantics, Algorithms and Tools, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
6. M. Fowler. *Domain-specific Languages*. Pearson Education, 2010.
7. C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing Non-functional Uncertainty via Model-driven Adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 33–42, Piscataway, NJ, USA, 2013. IEEE Press.
8. K. Havelund et al. *Formal Methods for Real-Time and Probabilistic Systems*, chapter Formal Verification of a Power Controller Using the Real-Time Model Checker Uppaal, pages 277–298. Springer.
9. A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, chapter Testing Real-Time Systems Using Uppaal, pages 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
10. M. U. Iftikhar and D. Weyns. ActivFORMS: Active FORmal Models for Self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 125–134, NY, USA, 2014. ACM.
11. D. G. D. L. Iglesia and D. Weyns. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Trans. Auton. Adapt. Syst.*, 10(3):15:1–15:31, Sept. 2015.
12. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov 1990.
13. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Computer aided verification*, pages 585–591. Springer, 2011.
14. Y. Laurent, R. Bendraou, S. Baarir, and M.-P. Gervais. Formalization of fUML: An Application to Process Verification. In *Advanced Information Systems Engineering*, pages 347–363. Springer, 2014.
15. S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
16. T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(K) Parser Generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995.
17. D. C. Schmidt. Model-driven Engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.
18. S. Shevtsov, M. U. Iftikhar, and D. Weyns. SimCA vs ActivFORMS: Comparing Control- and Architecture-based Adaptation on the TAS Exemplar. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, CTSE 2015, pages 1–8, New York, NY, USA, 2015. ACM.
19. M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, Bibliothek der RWTH Aachen, 2000.